

Have you checked out
EE Times DESIGN LIBRARY?

Embedded.com

Firmware basics for the boss

By Jack Ganssle, Embedded Systems Programming

Jan 22 2004 (13:00 PM)

URL: <http://www.embedded.com/showArticle.jhtml?articleID=17500630>

If your boss understands the beast that is firmware development, your team can be more productive. Here's what the boss needs to know.

I hear from plenty of readers that their bosses just don't "get" software. Efforts to institute even limited methods to produce better code are thwarted by well-meaning but uninformed managers chanting the "can't you just write more code?" mantra.

Yet when I talk to the bosses many admit they simply don't know the rules of the game. Software engineering isn't like building widgets or designing circuit boards. The disciplines are quite different, the techniques and tools vary, and the people themselves all too often quirky and resistant to standard management ploys. Most haven't the time or patience to study dry tomes or keep up with the standard journals. So this month and next I'm doing a short intro to the subject. Here's the first installment; give it to your boss.

Firmware costs

So, dear boss, assuming you're reading this, the first message is one you already know. Firmware is the most expensive thing in the universe. Building embedded code will burn through your engineering budget at a rate matched only by a young gold-digger enjoying her barely sentient ancient billionaire's fortune.

Most commercial firmware costs around \$15 to \$30 per line, measured from the start of a project till it's shipped. When developers tell you they can "code that puppy over the weekend," be very afraid. When they estimate \$5/line, they're on drugs or not thinking clearly. Defense work with its attendant reams of documentation might run upwards of \$100 per line or more; the space shuttle code is closer to \$1,000 per line, but is without a doubt the best code ever written.

The rate of \$15 to \$30 per line translates into a six-figure budget for even a tiny 5k line application. The moral: embarking on any development endeavor without a clear strategy is a sure path to squandering vast sums.

Like the company that asked me to evaluate a project that was five years late and looked more hopeless by the day. I recommended they trash the \$40 million effort and start over, which they did. Or the startup that, despite my best efforts to convince them otherwise, believed the consultants' insanely optimistic schedule. They're now out of business—the startup, that is. The consultants are thriving.

Version control

First, before even thinking about building any sort of software, install and have your people use a version control system (VCS). Building even the smallest project without a VCS is a waste of time

and an exercise in futility.

The NEAR spacecraft dumped a great deal of its fuel and was nearly lost when an accelerometer transient caused the on-board firmware to execute abort code—incorrect abort code, software that had never really been tested. Two versions of the 1.11 flight software existed; unhappily, the wrong set flew. The code was maintained on uncontrolled servers. Anyone could, and did, change the software. Without adequate version control, it wasn't clear what made up correct shipping software.

A properly deployed VCS ensures these sorts of dumb mistakes just don't happen. The VCS is a sort of database for software, releasing the code to users but tracking who changed what when. Why did the latest set of changes break working code? The VCS will report what changed, who did it, and when, giving the team a chance to efficiently troubleshoot things.

Maybe you're shipping release 2.34, but one user desperately requires the old 2.1 software. Perhaps a bug snuck in sometime in the last 10 versions and you need to know which code is safe. A VCS reconstructs any version at any time.

Have you ever misplaced code? In October 1999 the FAA announced they had lost the source code to all of the software that controlled air traffic between Chicago and the regional airports. The code all lived on one developer's machine, one angry person who quit and deleted it all. He did, however, install it on his home computer, encrypted. The FBI spent six months reverse engineering the encryption key to get their code back. Sound like disciplined software development? Maybe not.

Without a VCS, a failure of any engineer's computer will mean you lose code, since it's all inevitably scattered around amongst the development team. Theft or a fire—unhappily everyday occurrences in the real world—might bankrupt you. The computers have little value, but that source code is worth millions.

The version control database—the central repository of all of your valuable software—lives on a single server. Daily backups of that machine, stored offsite, ensures your business's survival despite almost any calamity.

Some developers complain that the VCS won't protect them from lazy programmers who cheat the system. You or your team lead should audit the VCS's logs occasionally to be sure developers aren't checking out modules and leaving them on their own computers. A report that takes just seconds to produce will tell you who hasn't checked in code, and how long it has been out on their own computers.

Version control systems range in price from free (like the GNU products) to expensive, but even the expensive ones are cheap. For a comprehensive list of products, see www.codeorganizer.com/version_control/tools.htm.

Firmware standards

Although English is spoken throughout America, try talking to random strangers on a street corner in Baltimore today. The dialects range from educated middle-American to incomprehensible dialects. It's all English, of a sort, but it sounds more like the fallout from the Tower of Babel.

In the firmware world we speak a common language: C, C++, or assembly, usually. Yet there's no common dialect; developers exploit different aspects of the lingo or create their programs using legal but confusing constructs.

The purpose of software is to work, of course, but also to clearly communicate the programmer's intentions to maintenance people. Clear communication means we must all use similar dialects.

Someone—that's you, boss—must specify the dialect.

The C and C++ languages are so conducive to abuse that there's a yearly obfuscated C contest whose goal is to produce utterly obscure but working code. Normally I don't publish the URL as these people are code terrorists who should be hunted down and shot like the animals they are, but the examples are compellingly illustrative. To see how bad things can get, see www0.us.ioccc.org/2001/williams.c. And then vow that your group will produce world-class software that's cheap to maintain.

The code won't be readable unless we use constructs that don't cause our eyes to trip and stumble over unusual indentation, brace placement, and the like. That means setting rules, a standard, used to guide the creation of all new code.

The standard defines far more than stylistic issues. Deeply nested conditionals, for instance, lead to far more testing permutations than any normal person can manage. So the standard limits nesting. It specifies naming conventions for variables, promoting identifiers that have real meaning. Tired of seeing **i**, **ii**, and (my personal favorite) **iii** for loop variable names? The standard outlaws such lazy practices. Rules define how to construct useful comments. Comments are an integral and essential part of the source code, every bit as important as for and while loops. Replace or retrain any team member who claims to write "self commenting code."

Some developers use the excuse that it's too time consuming to produce a standard. Plenty exist on the net; mine is in Word doc format at www.ganssle.com/misc/fsm.doc. It contains the brace placement rule that infuriates the most people—you'll change it and make it your own.

So write or get a firmware standard. And boss, please work with your folks to make sure all new code follows the standard.

Code inspections

What's the cheapest way to get rid of bugs? The answer's simple: just don't put any in!

Trite, perhaps, yet there's more than a grain of wisdom there. Too many developers crank lots of code fast and then spend ages fixing their mistakes. The average project eats 50% of the schedule in debugging and test. Reduce debugging by inserting fewer bugs, and you'll accelerate the schedule.

Inspect all new code. That is, use a formal process that puts every function in front of a group of developers before they spend any time debugging. The best inspections use a team of about four people who examine every line of C in detail. They'll find most of the bugs before testing.

Study after study shows inspections are 20 times cheaper at eliminating bugs than debugging. Maybe you're suspicious of the numbers—fine, divide by an order of magnitude. Inspections still shine, cutting debugging in half.

More compellingly it turns out that most debugging strategies never check half the code. Things like deeply-nested IF statements and exception handlers are tough to test. My collection of embedded disasters shows a similar disturbing pattern: most stem from poorly executed, pretty much untested error handlers.

Inspections and firmware standards go hand in hand. Neither works without the other. The inspections ensure programmers code to the standard, and the standard eliminates inspection-time arguments over stylistic issues. If the code meets the standard, then no debates about software styles are permitted.

Most developers hate inspections. Tough. You'll hear complaints that they take too long. Wrong. Well-paced inspection meetings examine 150 lines of code per hour, a rate that's hardly difficult to maintain (that's 2.5 lines of C per minute), yet that costs the company only a buck or so per line. Assuming, of course, that the inspection has no value at all, which we know is simply not true.

Your role, boss, is to grease the skids so the team efficiently cranks out fabulous software. Inspections are a vital part of that process. They won't replace debugging, but will find most of the bugs very cheaply.

Have your people look into inspections closely. The classic reference is *Software Inspection* by Gilb and Graham, but Karl Wiegers' newer and much more readable book *Peer Reviews in Software: A Practical Guide* targets teams of all sizes (including solo programmers).^{1, 2}

Toss out bad code

A little bit of the software is responsible for most of the debugging headaches. When your developers are afraid to make the smallest change to a module, that's a sure sign it's time to rewrite the offending code.

Developers tend to accept their mistakes, to attempt to beat lousy code into submission. It's a waste of time and energy. Barry Boehm showed in *Software Engineering Economics* that the crummy modules consume four times the development effort of any other module.³

Identify bad sections early, before wasting too much time on them, and then recode. Count bug rates using bug tracking software. Histogram the numbers occasionally to find those functions whose error rates scream "fix me!" and have the team recode.

Figure on tossing out about 5% of the system. Remember that Boehm showed this is much cheaper than trying to fix it.

Don't beat your folks up for the occasional function that's a bloody mess. They may have screwed up, but have learned a lot about what should have been done. Use the experience as a chance to create a killer implementation of the function, now that the issues are clearly understood. Healthy teams use mistakes as learning experiences.

Use bug tracking software, such as the free bugzilla (www.bugzilla.org/) or any of dozens of commercial products (nice list at www.aptest.com/resources.html).

Even the most disciplined developers sometimes do horrible things in the last few weeks to get the device out the door. Though no one condones these actions, fact is that quick hacks happen in the mad rush to ship. That's life. It's also death for software.

Quick hacks tend to accumulate. Version 1.0 is pretty clean, but the evil inflicted in the last few weeks of the project add to problems induced in 1.1, multiplied by an ever-increasing series of hacks added to every release. Pretty soon the programming team says things like "we can't maintain this junk anymore." Then it's too late to take corrective action.

Acknowledge that some horrible things happened in the shipping mania. But before adding features or fixing bugs in the next release, give the developers time to clean up the mess. Pay back the technical debt they incurred in the previous version's end game. Otherwise these hacks will haunt the system forever, reduce overall productivity as the team struggles with the lousy code in each maintenance cycle, and eventually cause the code to rot to the point of uselessness.

Stay tuned for more next month.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.

Further reading

1. Gilb, Tom and Dorothy Graham. *Software Inspection*. Addison-Wesley: NY NY, 1993.
 2. Wiegers, Karl. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley: NY Boehm, Barry. *Software Engineering Economics*. Prentice Hall PTR: Upper Saddle River, NJ, 1981.
-

Excellent article. As the only software engineer in the company, even inspecting a set of code you haven't looked at in a while is a huge help. When writing new code, I've found that writing it and then testing it in one day and then inspecting it the next morning works. Thanks again.

- Dan Stone

Next month, please include tips for bosses who used to be software engineers. Such managers naturally profess to understand our jobs, yet the schedule still reigns supreme.

- David M. Tomer

Jack replies:

David, all the rules are the same. I'll just add one more: Be Brave. Stand up for what you know is right. Don't cave to unrealistic pressures.

I like a lot of what you said in this article, but I wanted to add a correlary to your comments on code inspections. To have code inspections on at most 150 lines of code and to have them every time you want to submit fixes, I think its essential to reduce the overhead of the inspections.

On my project, the bureaucratic overhead of inspections are awful. I can literally spend a whole day pulling together the code packets making sure all of my SQA boxes are checked. Another several hours of effort are required to make sure all of the SQA forms are completed after the review to make sure all of the forms are completed.

This overhead kills. What happens is people stop following the 150 line maximum rule. I've gone to reviews where people have literal pages of software to inspect. Nobody comes prepared to such a review.

Or, nobody wants to make changes because they don't want to have an inspection. Software development and especially innovation slows to a halt. We are stuck with bad and bulky software because nobody wants to refactor.

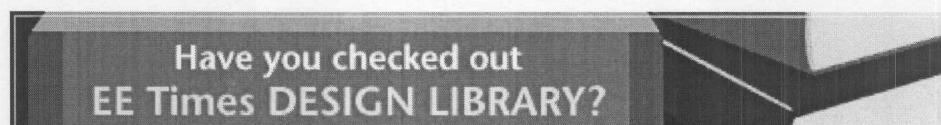
Finally, who you invite to the review is critical. You want people to attend that know the code. I've gotten little valuable input from people who were required to attend the review but who have little knowledge on the working software.

Maybe these issues are covered in the books you reference. I'll have to check that out.

Scott, it sounds like your problem isn't the code inspection, but the paperwork you have to fill out to prove you did the inspection. Is this some sort of government project? I spent a few years in defense contracting (hardware, not software), and documenting that we'd done the job right cost about twice as much as doing the job right...

- Mark Moss

Copyright 2003 © [CMP Media LLC](#)



Cut the Cord the Easy Way.



Win a FREE Dev Kit (\$495 value).



CYPRESS

Embedded.com

Firmware basics for the boss, part 2

By Jack Ganssle, Embedded Systems Programming

Feb 5 2004 (14:00 PM)

URL: <http://www.embedded.com/showArticle.jhtml?articleID=17602061>

As the boss, you can get the best out of your engineers by helping them hone their craft—the key to your mutual success.

Firmware engineers often tell me they want to do the right things, such as employing careful design practices but are defeated by their bosses' ignorance of software issues. Sound familiar? If so, cut out last month's and this month's column and slip them under your boss's door. It will give the boss the tools he or she needs to be more successful.

Tools

A poll on [Embedded.com](http://www.embedded.com) suggests 85% of companies won't spend more than \$1,000 on any but the most essential tools. Considering the \$100k+ loaded cost of a single engineer, it's nuts to not spend a few grand on a tool that offers even a small productivity boost.

Like what? Lint, for one. Lint is a program that examines the source code and identifies suspicious areas. It's like a compiler's syntax checker on steroids. Only Lint is smart enough to watch variable and function usage across multiple files. Compilers can't do that. For a fraction of the cost, aggressive Lint usage picks out many problems before debugging starts. Lint all source files before doing code inspections.

Gimpel (www.gimpel.com) sells one for \$239. It's up to you to buy it and to ensure your engineers use it on all new code. Lint is annoying at first, often initially zeroing in on constructs that are indeed fine. Don't let that quirk turn your people off. Tame it, and then reap great reductions in debugging times.

Debugging eats 50% of most projects' schedules. The average developer has a 5 to 10% error rate. Anything that trims that rate even a smidgen saves big bucks.

Make sure the developers aren't cheating their tools. Warning levels on compilers, for instance, should be set to the lowest possible level so all warnings are displayed. And then insist the team write warning-free code. It's astonishing how we ship firmware that spews warnings when compiled. The compiler, which understands the language's syntax far better than any of your people, is in effect shouting "Look here. Here! This is scary!" How can anyone ignore such a compelling danger sign?

Write warning-free code so that maintenance people in months or decades won't be baffled by the messages. "Is it supposed to do this? Or did I reinstall the compiler incorrectly? Which of these is important?" This means changing the way they write C. Use explicit casting. Parentheses when there's any doubt. These are all good programming practices anyway, with zero cost in engineering, execution speed, or code size. What's the downside?

Editors, compilers, linkers, and debuggers are essential and non-negotiable tools as it's impossible to do any development without these. Consider others. Complexity analyzers can yield tremendous insight into functions, identifying "bad code" early before the team wastes their time and spirits trying to beat the cruddy code into submission. See www.chris-lott.org/resources/cmetrics for a list of freebies. Bug tracking software helps identify problem areas—see a list of resources at www.aptest.com/resources.html.

Most firmware developers are desperate for better debugging tools. Unhappily, the grand old days of in-circuit emulators are over. These tools provided deep insight into the intrinsically hard-to-probe embedded system. Their replacement, the background debug mode (BDM), offers far less capability. Have mercy on your folks and insist the hardware team dedicate a couple of spare parallel output bits just to the software people. They'll use these along with instrumented code for a myriad of debugging tasks, especially for hard-to-measure performance issues.

Peopleware

Your developers—not tools, not widgets, not components—are your prime resource. As one wag noted, "my inventory walks out the door each night."

I've recommended several books in these two articles. Please, though, read *Peopleware* by DeMarco and Lister.¹ It's a slender volume that you'll plow through in just a couple of enjoyable hours. Pursuing the elusive underpinnings of software productivity, for 10 years the authors conducted a "coding war" between volunteering companies.

The results? Well, at first the data was a scrambled mess. Nothing correlated. Teams that excelled on the projects (by any measure: speed, bug count, matching specs) were neither more highly paid nor more experienced than the losers. Crunching every parameter revealed the answer—developers imprisoned in noisy cubicles, those who had no defense against frequent interruptions, did poorly.

How poorly? The numbers are breathtaking. The best quartile was 300% more productive than the lowest 25%. Yet privacy was the only difference between the groups.

Think about it—would you like three times faster development?

It takes your developers 15 minutes, on average, to move from active perception of the office busyness to being totally and productively engaged in the cyberworld of coding. Yet a mere 11 minutes passes between interruptions for the average developer. Ever wonder why firmware costs so much? E-mail, the phone, people looking for coffee filters, and sometimes you, boss, all clamor for attention.

Sadly, most developers live in cubicles today, which are, as Dilbert so astutely noted, "anti-productivity pods." Next time you hire someone, peer into his cube occasionally. At first he's anxious to work hard, focus, and crank out a great product. He'll try to tune out the poor bloke in the next cube who's jabbering on the phone with his lawyer about the divorce. But we're all human; after a week or so he's leaning back from the keyboard, ears raised to get the latest developments. A productive environment? Nope.

I advise you to put your developers in private offices, with doors and off-switches on the phones. You probably won't do that. Every time I've fought this battle with management I've lost, usually because the interior designers promise cubes offer more "flexibility." But even cubicles have options.

Encourage your people to identify their most productive hours—that time of day when their brains are engaged and working at max efficiency. Me, I'm a morning person. Others have different

habits. But find those productive hours and help them shield themselves from interruptions for about three hours a day. In that short time, with the three-times productivity boost, they'll get an entire day's work done. The other five hours can be used for meetings, e-mail, phone contacts, supporting other projects, and so on.

Give your folks a curtain to pull across the cube's opening. Obviously a curtain rod would decapitate employees, generally a bad idea despite the legions of unemployed engineers clamoring for work. Instead, use a Velcro strip to secure the curtain in place. Put a sign on the curtain labeled "enter and die"; the sign and curtain go up during the employee's three superprogramming hours per day. Train the team to respect their colleagues' privacy during these quiet hours. At first they'll be frantic: "but I've GOT to know the input parameters to this function or I'm stuck!" With time they'll learn when Joe, Mary, or Bob will be busy and plan ahead. Similarly, if you really need a project update and Shirley has her curtain up, back slowly and quietly away. Wait till their hours of silence are over.

Have them turn off their phone during this time. If Mary's spouse needs her to pick up milk on the way home, well, that's perfect voicemail fodder. If the kids are in the hospital, then the phone attendant can break in on her quiet time.

The study took place before e-mail was common. You know, that cute little bleep that alerts you to the same tired old joke that's been circulating around the 'net for the last three months, while diverting attention from the problem at hand. Every few seconds, it seems. Tell your people to disable e-mail while cloistered.

When I talk to developers about the interruption curse they complain that the boss is the worst offender. Resist the temptation to interrupt. Remember just how productive that person is at the moment and wait till the curtain comes down.

(If you're afraid the employee is hiding behind the curtain surfing the net or playing Doom, well, there are far more severe problems than just productivity issues. Without trust—mutual trust—any engineering department is in trouble).

Remember the Titanic

Where should you use your best people? It's natural to put the superprogrammers on the biggest and most complex projects. Resist that urge—it's wrong.

Capers Jones, in a private study produced for IBM, showed that the best people excel on small (one man-month) projects, typically being six times more productive than the worst members of the team.² That advantage diminishes as the system grows. On an eight man-month effort the ratio shrinks to under 3 to 1. At 64 man-months it's about 1.5 to 1, and much beyond that the best do as badly as the worst. Or the worst as well as the best. Whatever.

That observation tells us something important about how we partition big projects. Find ways to break big systems down into many small, mostly independent parts. Or at least strip out as much as possible from the huge carcass of code you're planning to generate, putting the removed sections into their own tasks or even separate processors. Give these smaller sections to the superprogrammers. They'll crank out solutions fast.

For example, suppose an I/O device, say an optical encoder, is tied to your system. Remove it. Add a CPU, a cheap PIC, AVR, Z8, or similar sub-one-dollar part, just to manage that one device. Have it return its data in engineering units: "the shaft angle is 27 degrees." Even a slowly rotating encoder would generate thousands of interrupts a second, a burden to even the fastest CPU that's also tasked with many other activities. Yet even a tiny microcontroller can easily handle the data if there's nothing else going on. One smart developer can crank out perfect I/O code in little time.

(An important rule of thumb states that 90% loaded systems double development time, compared to one of 70% or less; 95% loading triples development time.)

While cleverly partitioning the project for the sake of accelerating the development schedule, think like the customer does, not as the firmware folks do. The customer only sees features; never objects, ISRs, or functions. Features are what sell the product.

That means break the development effort down into feature-chunks. The first feature of all, of course, is a simple skeleton that sets up the peripherals and gets to **main()**. That and a few critical ISRs, perhaps an RTOS, and the like form the backbone upon which everything else is built.

Beyond the backbone are the things the customer will see. In a digital camera there's a handler for the charged-couple device, a liquid crystal display subsystem, some sort of flash filesystem. Cool tricks like image enhancement, digital zoom, and much more will be the sizzle that excites marketing. None of those, of course, has much to do with the basic camera functionality.

Create a list of the features and prioritize. What's most important? Least? Then—and this is the trick—implement the most important features first.

Does that sound trite? It is, yet every time I look at a product in trouble no one has taken this step. Developers have virtually every feature half-implemented. The ship date arrives and nothing works. Worse, there's no clear recovery strategy since so much effort has been expended on things that are not terribly important.

So in a panic, management starts tossing out features. One 2002 study showed that 74% of projects wind up with 30% or more of the features being eliminated. Not only is that a terrible waste—these are partially implemented features—but the product goes to market late, with a subset of its functionality. If the system were built as I'm recommending, even schedule slippages would, at worst, result in scrubbing a few requirements that had as yet not consumed engineering time. Failure, sure, but failure in a rather successful way.

Dollars and sense

Finally, did you know great code—the really good stuff, that which has the highest reliability—costs the same as cruddy software to develop? This goes against common sense. Of course, all things being equal, highly safety-critical code is always much more expensive than consumer-quality code.

But what if we don't hold all things equal? A study by Oddur Benediktsson showed that using higher and higher levels of disciplined software process lets one build higher-rel software at a constant cost.³ If your projects march from low reliability along an upwards line to truly safety-critical code, and if your outfit follows, in his study, increasing levels of the Capability Maturity Model, the cost remains constant.

Makes one think. And hopefully, it makes one rein in the hackers who are more focused on cranking code than specifying, designing, and carefully implementing a world-class product.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.

References

1. DeMarco, Tom and Timothy Lister. *Peopleware—Productive Projects and Teams*, 2nd Edition.

New York, NY: Dorset House Publishing, 1999.

2. Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Yourdon Press, 1993.
3. Benediktsson, Oddur. *Safety Critical Software and Development Productivity*, conference proceedings, Second World Conference on Software Quality, September 2000.

After reading the first two parts of your series, I don't know whether to laugh or cry! The whole issue of interruptions has been something that I have tried to deal with for many years, and so far have not been able to resolve. Here, I am the only person who does electronic product design. That means concept, circuit design, PC board design, software, packaging, occasionally instructions, production support, customer support (occasionally) and, when I have nothing to do, R&D!

I have tried to get additional design personnel, tech support people, production engineers, etc. over the years, but we have been hampered by lack of money and difficulty in finding people with the right combination of skills & personality. Part of this is related to the lack of money issue!

At any rate - I am a poster child for the problems that occur from rapid context switching. I have sometimes reached the point where I sit at my desk (and I do enjoy the luxury of my own office) and literally do nothing because I can't figure out which of a dozen things to do needs my attention the most. I ask for priorities so that I can focus on projects that are most necessary, but the priority can change in a few minutes, depending upon what strikes the boss' fancy at the time.

My advice to managers is to allow your people to choose the tools they need to get the work done, and make sure that YOU have your priorities in order & back up your people when they tell some other manager or department, "No, I can't do that right now."

OK, venting over! Back to the grind!

- Dave Telling

Firmware costs. The issue with delays in schedule and over commitment, in my mind, are mainly due to having inexperienced software engineers and/or a poor specification or scope. Trying to prepare Sr. managers for "gold-digger" engineers just is not feasible. If I spent \$40,000,000 on a project that was scrapped, I would hope someone would fire me. Somewhere around 2 years later, I hope I would have replaced my staff. If time and money aren't the problem, the logical conclusion is scope. The guy that tells me he can "code that puppy over the weekend" and does, is the one I give a raise to.

In general, my problem with your article is that it simply talks about control. In my humble opinion, bosses should not have to control the process. Software engineers need to realize that the work is just a little bit of magic. The magic they produce is vital to the survival of their company. If they don't get it done, the project, and even the company could fail. Excuses, delays, not using VCSs, bug tracking, and skipping reviews should not be tolerated. There are lots of very good programmers without jobs today because of those things. The countless hours of planning, lobbying for money, and creating the project scope are indeed the jobs of the bosses; execution is in the hand of the engineer.

- Keith Hansen

Copyright 2003 © CMP Media LLC